UML Specifications Toward a Codesign Environment

Marcello Lajolo
NEC Laboratories America
Princeton NJ 08540, USA

Ananda Shankar Basu, Mauro Prevostini
ALaRI, University of Lugano
Lugano, Switzerland

Abstract

The Unified Modeling Language (UML) is receiving more and more attention from system designers that need to model both hardware and software related aspects of a system. On the ground of the growing consensus toward the need to raise the level of abstraction in system specifications, we would like to present a methodology that aims to address embedded systems design issues at multiple levels of abstraction and to support a function/architecture codesign process. Our approach integrates UML with high-level synthesis and hardware/software co-verification techniques in order to provide an automated flow for SoC design starting from system-level specifications down to hardware/software partitioning and integration. UML has been selected because it is platform independent and helps team members to share very efficiently relevant information during the various design phases, while high-level synthesis helps to evaluate constraints that the embedded system must satisfy: e.g. performance, power and cost starting from behavioral specifications.

1 Introduction

With the increasing design complexity and the reduction of the time-to-market windows, the design of electronic systems has become a challenging task to be handled by traditional methodologies. Embedded systems design in comparison to traditional software development requires not only to verify the functional correctness, but also to satisfy tight performance and cost constraints. Hence, new methodologies are needed to improve design productivity and derive high-performance low-cost implementations keeping in mind the reuse of pre-designed components.

The software community, after several years of work, converged on a set of notations for developing specifications of object-oriented systems known as the Unified Modeling Language or UML [RJB98] that has been very successful as a visual way for describing software. However, UML is not limited to software modeling and the development of UML 2.0 has been undertaken with the express intention of producing a language that has benefits for a much wider audience than just software developers, including the world of systems engineering.

In this work, we present an integration of a UML-based modeling methodology with a C-based design technology called Aces (**A**pplication to **C** to **E**xploration to **S**ystem LSI) [Laj03] that leverages on high-level synthesis and co-verification tools and aims to assist the designer in the hardware/software partitioning and architecture selection phases. Aces has the unique advantage with respect to all similar approaches to be able to leverage off the strengths of two key pieces in NEC's C-based design flow [WO00]: Cyber, a behavioral hardware synthesis tool and Classmate, a hardware/software co-verification tool. UML complements Aces with an object oriented modeling language with both graphical and textual notations, organized in a set of diagrams, each diagram capturing a different aspect, or level of abstraction, of the system. The result is a unified design flow from system specification down to system implementation.

This paper is organized as follows. The state-of-the-art about SoC with UML is presented in Section 2, while Section 3 shows the proposed flow and the Co-design environment used in this methodology. Section 4 describes in detail, with an example case study, our methodology for SoC design starting from UML specification and in Section 5 you will find our conclusions.

2 State of the art and contribution

Electronic System Level (ESL) design methodologies are receiving more and more attention from Electronic Design Automation (EDA) vendors. For example, commercial hardware and software co-verification tools from companies such as Mentor Graphics, CoWare, VAST, Virtio and Axys can provide fast instruction-set simulators linked to various hardware simulators. They mainly focus on the functional and performance modeling problem for software-dominated embedded systems, although they do not address the issues of high-level hardware modeling and refinement. The main limitation of these tools is that they often require to model the hardware at the RT-level and even though recently some of these vendors have started to offer the possibility to perform a mix C/RTL co-verification (e.g. C-Bridge from Mentor Graphics), none of them offers yet an automated behavioral synthesis path from behavioral specifications.

An emerging area is also the one of coprocessor synthesis [Men, Syn, Cri], where the main idea is to combine the software compilation and the hardware synthesis technologies to provide a system that allows designers to explore and implement their designs directly from descriptions written in algorithmic C. The main limitation of this approach is that it is based on the assumption that the designer has already been able to come up with a feasible hardware/software partitioning for the entire design and the coprocessor synthesizer can then provide the possibility to perform some software acceleration by off-loading compute-intensive algorithms from the CPU to dedicated hardware. Although very useful, tools of this type can only provide a partial support to a complete SoC design flow because it is well known that many decisions regarding the efficiency (performance, power, area etc.) of the system have largely been fixed by the time a designer commits to a particular architecture.

Alternative and complementary methodologies and solutions must hence be provided in order to help the designer during the initial phases of the design process when coarse hardware/software partitioning trade-offs have to be analyzed. Our work is an attempt to try to fill this gap by proposing a practical integration between a UML-based modeling methodology and an existing hardware/software codesign technology.

3 The Proposed Flow

The overall flow presented in this paper is shown in Figure 1. Our proposed methodology starts with the UML specification of the system, followed by exploration of the UML database for extraction of functional and structural information. This is followed by an interactive process performed through a web-based interface that allows to capture UML specifications and design constraints provided by the designer, like architectural specifications and hardware/software partitioning, and export the entire structure of the design into the ACES codesign environment. The following sections describe in detail the various phases in this design flow.

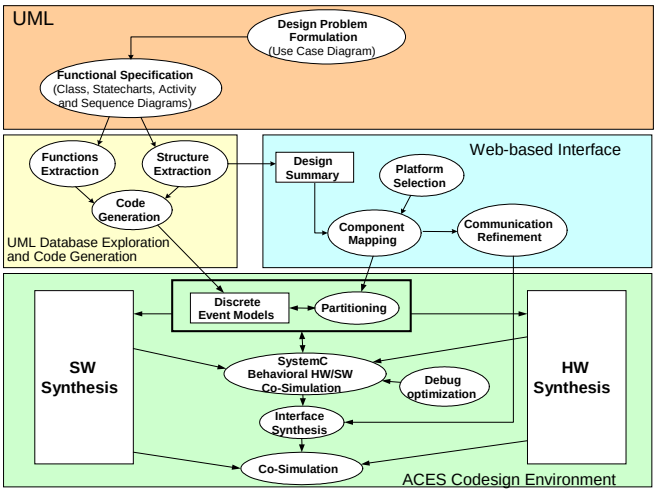

Figure 1: The overall design flow.

3.1 UML Specifications

The Unified Modeling Language (UML), is an object oriented modeling language that consists of graphical and textual notations, organized in a set of diagrams, each diagram capturing a different aspect or level of abstraction of the system [RJB98]. After getting the specifications of the system to be designed, the first step is to capture the functionality of the system as a whole using Use Case Diagrams. In the second step, the functionality is decomposed into components within Class (describing the SoC's structure), State Diagrams, Activity and Sequence Diagrams (describing the SoC's behavior). Constraints (i.e. timing) are captured using Stereotypes (simple extension mechanism of UML) and propagated and budgeted to the components. In the following step, the model is simulated in the UML environment in order to check whether the functional behavior of the system matches the original specifications.

For a first analysis of a possible integration between UML and codesign, we have started by considering a UML specification flow in which first an Object Model Diagram is defined to capture the structural decomposition into interacting components. An object model diagram contains two sets of classes: the ones whose behavior could potentially be implemented either in hardware and in software and others that do not have to enter in the codesign flow, for example, testbenches and strictly software oriented components. Classes belonging to the first set are distinguished by the ones of the second set using the *Partitionable* stereotype that has to be specified manually by the user (see [MMM+03]). Communication among classes can be specified through uni-directional relationships, associated to events, or by means of shared variables and we provide a specific API (further details are given in section 4.6) in order to guide the designer in this modeling phase. All partitionable classes are required to have a state diagram associated for specifying its run-time behavior, while non-partitionable classes may or may not have a state diagram associated.

As a next step, the UML Functional Specification must be translated into ACES Discrete Event Models to conjugate the convenience of using the graphical UML Platform interface for specification with the possibility to use the analysis and synthesis tools available using the ACES codesign methodology.

3.2 UML Database Exploration

The phase following the UML specification is that of extraction of functional and structural information from the UML database generating a textual summary of the UML specifications. This contains the

list of the modules(classes) and their interconnections(direct dependencies and shared variables).

3.3 Web-based interface

The next phase after the UML data base exploration makes use of a web-based interface and acts as an intermediate layer between UML and codesign. More information about this phase will be given in Section 4.7.

3.4 The ACES Codesign Flow

The back-end of the proposed methodology is the ACES codesign flow that is depicted in the bottom part of Figure 1. The system is described at the behavioral level as a network of discrete event models (tasks) that can communicate by both means of events as well as shared variables. Those models have a precise semantics and are written in SystemC. For each module in the system specification, ACES can synthesize a hardware netlist, a software program and the interfaces between hardware and software, based on partitioning and communication mapping information given manually by the user on a module by module basis. Behavioral SystemC co-simulation is used to test the behavior of the system and to perform hardware/software partitioning in a closed loop. Additional details regarding in particular software synthesis, interface generation and co-simulation within ACES are provided in [Laj03]. Good estimates of both hardware and software performance and power are of crucial importance in this phase in order to avoid costly design re-iterations. ACES provides the unique possibility to change the hardware/software implementation of each component in the system by simply changing an implementation parameter in the web browser. The same simulation code is used to simulate the functionality for both hardware and software implementations. The only things that change are the delay annotations that are used for modeling performance and power consumption and also the scheduling policy of the module in order to model shared system resources like the CPU.

4 From UML to Co-design

The link between UML specifications and an existing methodology for hardware/software codesign is the core of this paper. After the application is modeled and analyzed using the UML tool, we get a repository that contains information of the model in the internal database. We have used Rhapsody from I-Logix, Inc. as UML tool. We have found very useful the API's provided by Rhapsody to extract the information from the repository and generate the input files for the ACES environment. The transformation process has two phases:

1. code generation for synthesizable models, and
2. export of structural information

These two phases will be described in Sections 4.5 and 4.6, respectively, after having walked through an example of UML modeling.

4.1 The gfilter example

Let us start describing the *gfilter* application that is a small system that will allow us to see enough details about the overall methodology. The function of the system is to read a gray scale image stored

in an input memory, perform a filtering process based on iteratively substituting adjacent nine point squares with the average of their gray values and finally storing the filtered image in the output memory.

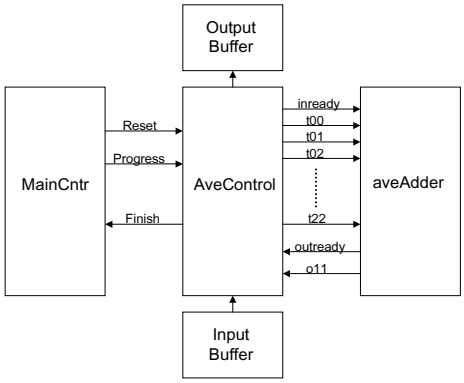

Figure 2: The gfilter problem description.

The specification of the system is as follows. AveControl is the only module that can access both the input and output memories. After having received the *progress* signal from the module MainCntr, it starts retrieving the input image, nine points at a time, from the input memory and passes them to the module aveAdder and signals the availability of new input data by raising the signal *inready*. The module aveAdder then computes the average and sends back the result to AveControl through the *o11* signal and raises the signal *outready*. At this point, AveControl starts a new computation. After the entire image has been processed, AveControl raises the signal *finish* that is received by the module MainCntr, and this determines the end of processing.

4.2 Object Model Diagram

Figure 3 describes the Object Model Diagram for our example model. This diagram shows the static structure of the specified system, in particular, classes, their internal structure including attributes, their methods and their relationships to other classes (such as inheritance or generalization and associations.)

Our system, in particular, is composed of three partitionable classes (MainCntr, AveControl, aveAdder) and one testbench (FileIO). Uni-directional relationships among classes are used in order to specify an event-based communication mechanism, while other data communications, such as the ones with the input and output memories have been implemented using global variables.

4.3 Sequence Diagram

After Use Cases and Object Model Diagrams have been developed, a Sequence Diagram can be specified as an additional form of interaction to help create testbenches. Figure 4 shows the exchange of signals and their sequence between the various modules in the gfilter system. Data communication across the modules through the function calls is shown in the figure.

4.4 State Diagrams

The next step is to create the state diagrams, that are descriptions based on Harel statecharts [Har87], used to model the behavior of each class in the system. The designer is responsible to figure out for each module what the states are, and how transitions happen between them. The transition indicates one

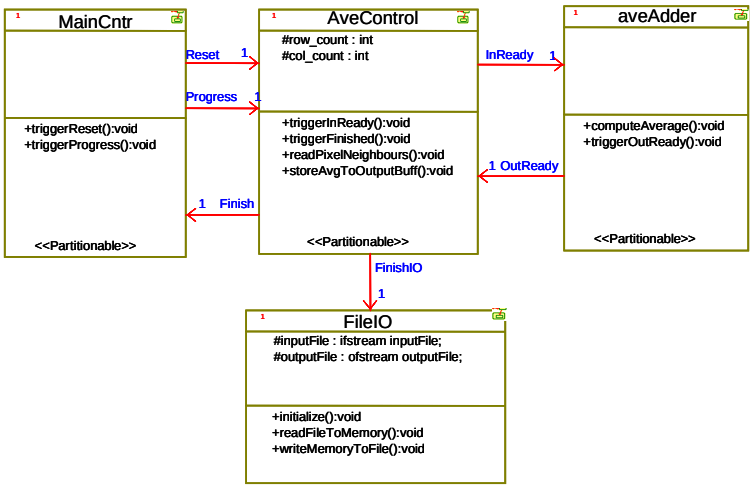

Figure 3: Object Model Diagram.

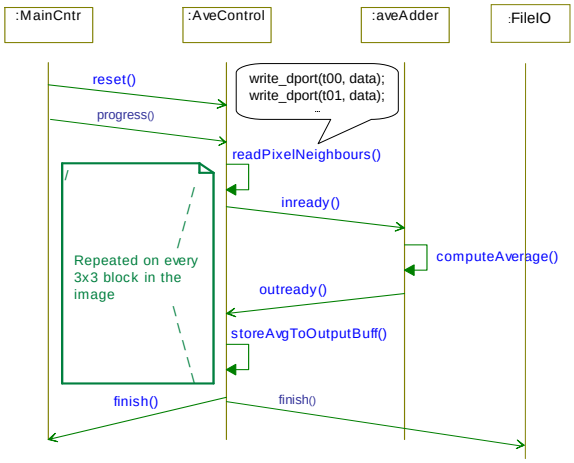

Figure 4: Sequence Diagram.

movement from one state to another. Each transition has a label that comes in three parts: `trigger-signature [guard]/activity`. All the parts are optional. States can also have some internal activity, like actions on entry and actions on exit, and there are some mechanisms to specify a delay for executing a transition. States can be broken into several orthogonal state diagrams that run concurrently and superstates can be used in order to share common transitions and internal activities among states. As an example, Figure 5 describes the state diagram for the AveControl class in the gfilter example. Here is shown an `AND` state containing a nested statechart with a `history` connector. An `AND` state is an orthogonal state which represents simultaneous independent substates that an object can be in the same time. A `history` connector stores the most recent active configuration of a state, so a transition to a history connector restores this configuration.

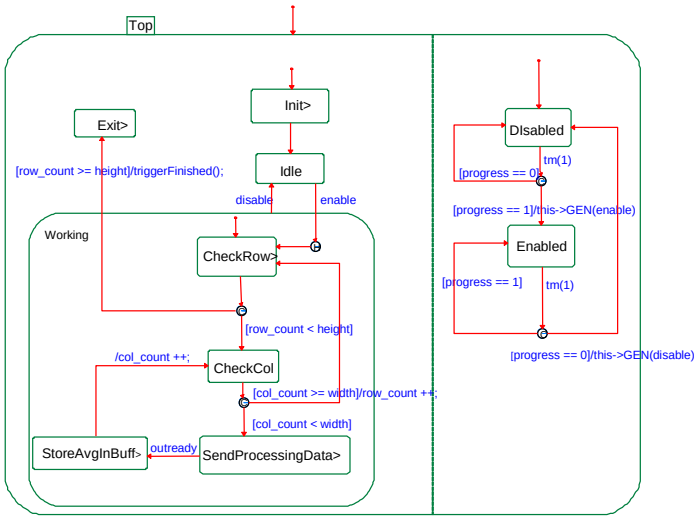

Figure 5: State diagram for module AveControl.

4.5 Code generation from state diagrams

The state diagrams describing the behavior of each partitionable class need to be converted into SystemC in order to be imported into the ACES codesign environment. From this textual representation, ACES is then able to perform both hardware and software synthesis. Figure 6 shows a portion of a SystemC description corresponding to the state diagram of Figure 5. We have chosen an unstructured style, due to its simplicity and efficiency, but many variants (e.g. nested switch, state pattern, state tables, etc.) are possible. Events are implemented as boolean terminals.

Figure 7 shows the pseudo-code of the algorithm that we have developed for automating this code generation process. The algorithm utilizes the Rhapsody's API in order to extract various information like list of classes, global variables and events in the model, the action/guard for the transitions, entry-action and exit-actions in a state, in transitions to a state, out transitions from a state, etc. Hence we can browse through the entire object model and extract relevant information regarding the state diagrams.

The algorithm is called on the root state of each state diagram for which code has to be generated. In every state, it first emits the code specified by the user in the action on entry portion of the state. Then it checks out-transitions from the state. For the transitions triggered by events, it issues a wait statement on that event, then it emits the code specified in the action on exit portion, followed by a goto statement, the label being the target state. In case of a conditional transition, it issues an if-then-else statement with goto labels depending on the condition. It also issues the code (if any) specified in the action section of the transition. Then the algorithm is called recursively on each state reachable by the current out transition. For an **AND** state, the same code generation algorithm is called on each of the sub-states within the **AND** state. The behavior is also the same for a state with nested statechart. This code generation step is actually controlled through the web-based interface that has been described in Section 4.7, since it requires the user to specify the hardware or software implementation for each partitionable module. We have implemented and tested the algorithm using Rhapsody UML tool, which provides API functions that allow us to extract all required information from a UML project database. However we would like to emphasize that this code generation algorithm is very general and can be utilized also with other UML tools.

```
#include <AveControl.h>

// External memories
extern sc_int<8> InputBuffer[height*width];
extern sc_int<8> OutputBuffer[height*width];

SC_MODULE(AveControl) {
  sc_in_clk clk;
  sc_in<bool> rst;

// Input terminals
  sc_in<sc_int<8>> o11;   // input data
  sc_in<bool> outready;   // input event */
  sc_in<bool> Reset;      //   "     "
  sc_in<bool> Progress;   // input event */

// Output terminals
  sc_out<bool> Finish;    // output event
  sc_out<bool> inready;   //   "      "
  sc_out<sc_int<8>> t00;  // output data
  … Omitted …

  SC_CTOR(AveControl) {
    SC_CTHREAD(main,clk.pos());
    watching(rst.delayed() == 0);
  }

void main(void) {
Init:
          row_count=0;
          goto Idle;
Idle:
          aces_wait(enable);
          goto Working;
Working:
    CheckRow:
              col_count=0;
              if ( row_count < height ) {
                          goto CheckCol;
              }
              else {
                          triggerFinish();   // Send Finish
                          goto Exit;
              }
    CheckCol:
       ... Omitted ...
  }
};
```

Figure 6: Code generated for AveControl.

4.6 Exporting structural information

In order to start with the codesign process, the last thing we need is to extract a summary of the design, essentially a textual representation containing a list of all the partitionable modules and their interconnections. In order to identify the partitionable modules, we require the user to specify the stereotype *Partitionable* on those modules that need to be considered in the co-design process.

UML allows to specify the description of a model through a wide variety of styles, but in order to perform a tight link with a codesign tool, we had to impose some restrictions to the user. In our system, the communication between partitionable entities can be described using events, data ports and shared variables. Events are a point-to-point communication mechanism used to describe the reactive behavior of a module and they generally trigger some transitions in a state diagram. Data ports are also a point-to-point communication mechanism, but they differ with respect to events because they do not trigger any transition. Their value can instead be used anywhere within the state diagram code. Finally, communication by means of shared variables is generally used in order to describe multiple access capabilities to a data that can be shared among different modules.

Our main contribution in this paper is to provide a specific API, basically an extended UML library, in order to allow the user to describe the type of communication that he wants to be performed. A summary of the macros provided with this API is shown in Table 1. These macros can be used in any portion of code used inside a state diagram (actions on entry state, actions on exit state, transition activity, etc).

Events are generated using the *event_gen* macro and can appear, as usual, in the trigger-signature of a transition. The event is associated to a uni-directional relationship (see arrows in Figure 3) that must

```
codeGenerate(state *S) {
  1. If S is visited, return; else mark S as visited.
  2. if 'S' is compound-state then
        for each sub-state 's' of 'S', codeGenerate(s)
  3. Issue code specified in the action-on-entry section (This code can be directly copied)
  4. Get out transitions {T}  from state S;
  5. {U} = empty;
  6. for each out-transition 't' of {T} do {
        if 't' is conditional {
           issue code specified in the action-on-exit section;
           s_t = target state if condition is true;
           s_f = target state if condition is false;
           issue if-then-else with goto label as 's_t' or 's_f' depending on condition;
           insert 's_t', 's_f' in {U};
        } else {
           s = target state of 't', insert 's' in {U};
           if 't' is triggered by event 'e' {
              issue wait on event 'e';
           }
           issue code specified in the action-on-exit section;
           issue goto with label as 's';
        }
        issue code specified in the action section of transition 't';
     }
     for each 'u' in {U} do
        codeGenerate(u);
}
```

Figure 7: Algorithm for code generation from a state diagram.

be created by the user in the class diagram. Read and write operations on a data port are described with the macros *read_dport* and *write_dport*. Read and write operations on a shared data are described with the macros *read_shared_data* and *write_shared_data*. For shared data we also provide three specific macros (*lock_shared*, *unlock_shared*, *check_shared_status*) in order to manage the mutual exclusiveness in the access to the data. Essentially, the internal implementation of a shared data makes use of a busy signal of type boolean. *lock_shared* prevents the use of a variable by other modules by setting this signal to 1, *unlock_shared* resets it and *check_shared_status* returns the value of the busy signal.

The internal implementation of this API is completely transparent to the user. We have implemented and tested it within the Rhapsody UML tool, but it can be supported in any other UML-based kind of technology. The macros of this API can be identified very easily during the exploration of the UML database of a project and allow us to export the information that we need for the following codesign phase.

API Macro Name	Description
event_gen(*relation_name,event_name*)	Generate an event through the specified relation instance
write_dport(*port_name,value*)	Write a value to a data port
read_dport(*port_name*)	Read from a data port
write_shared_data(*var_name,value*)	Write a value to a shared data
read_shared_data(*var_name*)	Read from a shared data
lock_shared(*var_name*)	Lock a shared data
unlock_shared(*var_name*)	Unlock a shared data
check_shared_status(*var_name*)	Check the status of a shared data

Table 1: Extended UML API.

4.7 Web-based interface

Figure 8 refers to the HTML page that is generated at the beginning of this phase. The two screen shots show the same page and respectively the top part (left side) and the bottom part (right side). This page can be opened using any web browser and is organized as follows. Starting from the top, there is a brief summary of the project containing its name and a short description. By clicking on a link, it is possible to see all the verbose report provided by the UML tool containing all the information about the project that has been collected in the UML database. The third line is used in order to select the platform onto which to implement the desired functionality. The selection is performed through a menu window where the user can pick any of the architectural templates available in a library provided with the codesign tool. An architectural template represents the platform for the system implementation and the user is responsible for selecting the platform that is best suitable for the system that he needs to implement (one or multiple CPUs, DSPs, simple or very complex bus hierarchy, etc).

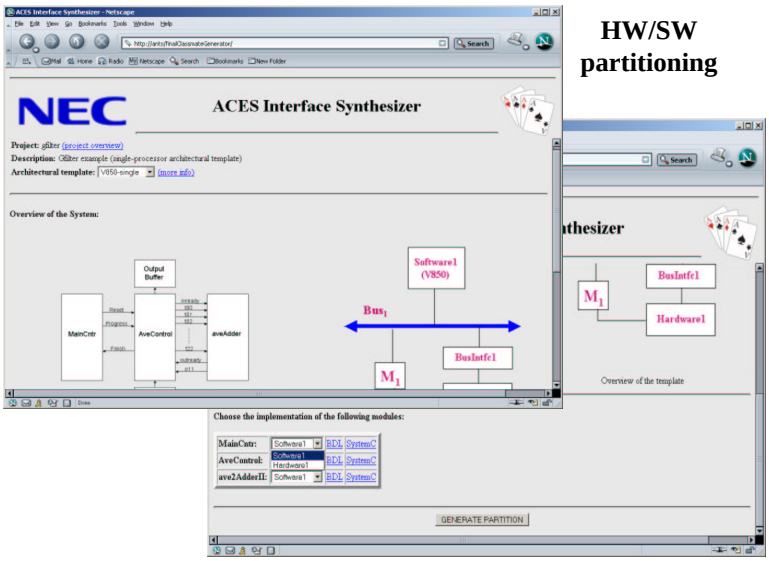

Figure 8: HTML page generated from UML specifications.

The selection of the platform is directly reflected in the graphical content, presented in the middle of the page, where on the left side there is the functional view of the system exported from the UML specifications and on the right side there is the picture of the selected platform. By changing the target platform, the picture on the right is automatically updated. For example in Figure 8 the platform contains only one processor, while in Figure 9 the platform contains two processors and a two-level bus hierarchy. The idea behind this solution is to support a function-architecture codesign approach that requires the separation of the functionality from the architecture selected for its implementation.

Finally, at the bottom of the page are listed all the modules present in the functional specifications and the user can specify the implementation (i.e., the hardware or software component of the platform onto which the functionality will be implemented.) for each of them through a menu window associated to each module. This is what we call *component mapping phase*. The number of choices available for this mapping depends on the selected platform. For example, in the platform shown in Figure 8, only two choices are possible (Software1, Hardware1) due to the fact that it is a simple single-processor architecture with one hardware component connected to the processor bus. But in the multi-processor architecture shown in Figure 9, five different choices are possible, since in this platform there are two processors and three hardware units.

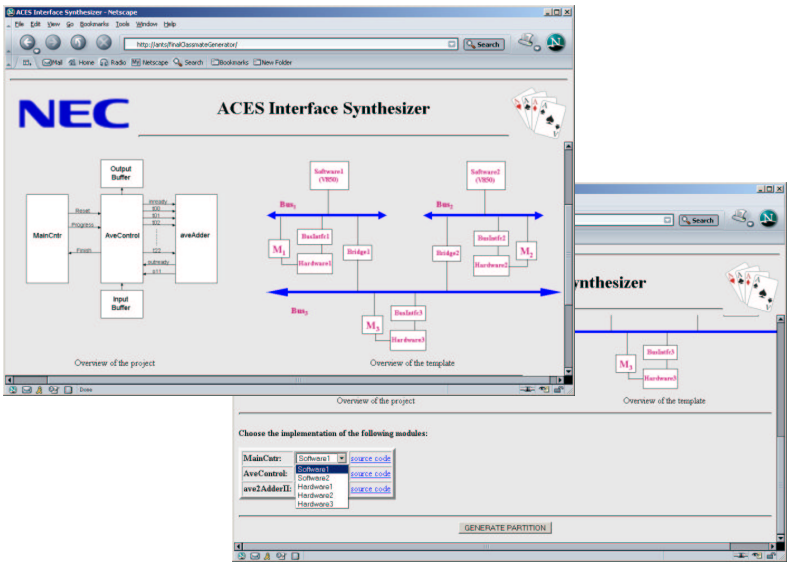

Figure 9: Mapping on a dual-processor architecture.

The component mapping phase ends when the user clicks the button "GENERATE PARTITION". This starts the process of analysis and characterization of all interface signals and opens a new html page, like the one shown in Figure 10, where all signals are listed.

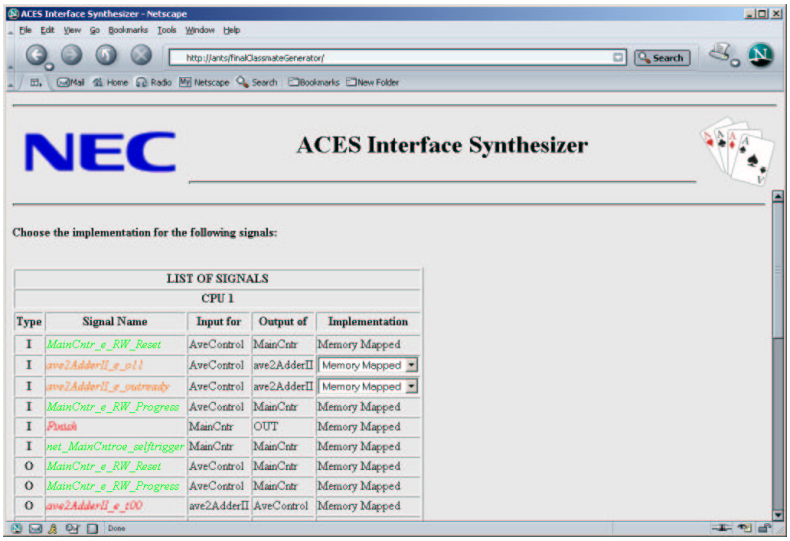

Figure 10: Communication refinement.

At this point, the *communication mapping* phase can start. The table shows all different types of connections: hardware-to-software, software-to-hardware, hardware-to-hardware and software-to-software connections. A connection can be recognized by its name, a color associated to its type, its source and its destination. The last column shows the specific implementation of the connection. Software-to-hardware connections are implemented in memory-mapped I/O, while hardware-to-software connections are by default implemented in memory-mapped, but the user can alternatively specify an interrupt-based implementation. Hardware-to-hardware signals are by default implemented as point-to-point communications, but the user can alternatively require the communication to be performed on the bus

(memory-mapped). Finally, software-to-software connections are implemented by the real-time operating system (RTOS). This list of signals presented in the table refers to a specific CPU in the selected platform and its associated system bus. In case of multi-processor platforms, several list of signals, one per CPU, are generated.

When all implementation options have been specified, the user can proceed by clicking the button "GENERATE CONNECTIONS", not shown in Figure 10, and at this point physical addresses will be generated for all memory-mapped communications and specific interrupt lines of the processor will be selected for signals implemented in interrupt. The result is a new page, not shown here, similar to Figure 10, but where the last column shows now the physical addresses and the interrupt lines that have been selected. After having examined all the communications, the user can still go back and change some implementation options or, if satisfied, proceed to the next hardware/software codesign phase.

5 Conclusions

The complexity of current embedded systems requires large teams of designers that interact especially at the early stages of the design when architecture selection and hardware/software partitioning take place. Models and tools that allow to visualize and document the design abstractions and the interactions between different components or levels of abstraction of a specification are essential. UML being platform independent and with a rich graphical notation can serve this purpose. We presented a methodology that specializes the UML standard notation for modeling embedded systems platforms and protocols leading to an integration with an existing hardware/software codesign technology.

References

[Cri] CriticalBlue:.
`http://www.criticalblue.com`.

[Har87] D. Harel. A visual formalism for complex systems. In *Science of Computer Programming*, 1987.

[Laj03] M. Lajolo. IP-Based SOC Design in a C-based design methodology. In *Proc. of IP Based SoC Design 2003*, pages 203–208, Oct. 2003.

[Men] Mentor's Application Specific Assistant Processor:.
`http://www.mentor.com/asap`.

[MMM+03] A. Minosi, S. Mankan, A. Martinola, F. Balzarini, A.N. Kostadinov, and M. Prevostini. UML-based Specifications of an Embedded Systems Oriented to HW/SW Partitioning: a Case Study. In *FDL'03 Proceedings*, pages 226–237, Sep. 2003.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[Syn] Synfora:.
`http://www.synfora.com`.

[WO00] K. Wakabayashi and T. Okamoto. C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective. *IEEE Trans. Computer-Aided Design*, 19(12):1507–1522, Dec. 2000.